Java Database Connectivity (JDBC)

Introduction

- JDBC (Java Database Connectivity) is defined, as a set of java classes and methods to interface with database.
- It also provides uniform access to a wide range of relational databases.
- The java 2 software bundle includes JDBC and the JDBC-ODBC Bridge.
- JDBC provides a standard API for tool/database developers and makes it possible to write database applications using a pure Java API.

What does JDBC do?

Simply put, JDBC makes it possible to do three things:

establish a connection with a database
send SQL statements
Process the results.

JDBC Is a Low-level API and a Base for Higher-level APIs

JDBC is a "low-level" interface, which means that it is used to invoke (or "call") SQL commands directly.
It works very well in this capacity and is easier to use than other database connectivity APIs, but it was designed also to be a base upon which to build higher-level interfaces and tools.
A higher-level interface is "user-friendly," using a more understandable or more convenient API that is translated behind the scenes into a low-level interface such as JDBC. At the time of this writing, two kinds of higher-level APIs are under development on top of JDBC:

1. **An embedded SQL for Java (Example SQLJ)**. At least one vendor plans to build this. DBMSs implement SQL, a language designed specifically for use with databases. JDBC requires that the SQL statements be passed as Strings to Java methods. An embedded SQL preprocessor allows a programmer to instead mix SQL statements directly with Java: for example, a Java variable can be used in a SQL statement to receive or provide SQL values. The embedded SQL preprocessor then translates this Java/SQL mix into Java with JDBC calls.
2. **A direct mapping of relational database tables to Java classes.** JavaSoft and others have announced plans to implement this. In this "object/relational" mapping, each row of the table becomes an instance of that class, and each column value corresponds to an attribute of that instance. Programmers can then operate directly on Java objects

ODBC API

Microsoft ODBC (Open Database connectivity) Application programming interface is probably the most widely used programming interface for accessing RDBMS.
ODBC is written entirely in C, JDBC is written in Java.
But, both JDBC and ODBC are based on the X/Open SQL Command level Interface.
It is an interface to perform SQL calls to database.
The SQL statements are embedded in the statements.

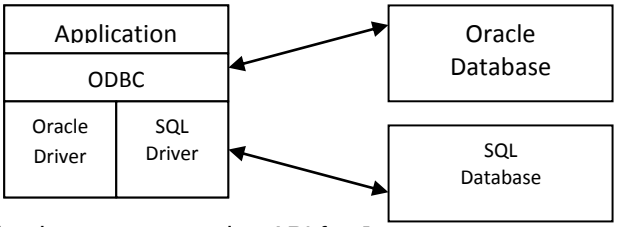

Fig. ODBC Database Design

JDBC API

JDBC provides a database programming API for Java programs.
Java programs cannot directly communicate with the ODBC driver.
Sun Microsystems provides a JDBC-ODBC bridge that translates JDBC to ODBC.
There are several type of JDBC drivers available they are:

- JDBC-ODBC bridge+ODBC driver
- Native API party java Driver
- JDBC-Net Pure java Driver
- Native protocol pure java Driver.

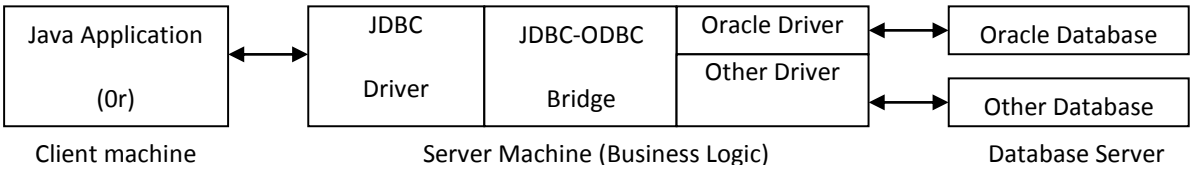

Fig. JDBC Application

JDBC-ODBC Bridge+ODBC driver (Type 1 Drivers)

These drivers are the bridge drivers such as the JDBC-ODBC Bridge.
These drivers rely on an intermediary such as ODBC to transfer the SQL calls to the database.
Bridge drivers often rely on native code, although the JDBC-ODBC library native code is part of the Java 2 virtual machine.

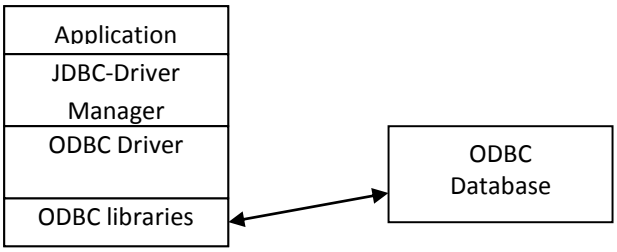

Fig. JDBC-ODBC Bridge Driver

Native API partly java Driver (Type 2 Drivers) (not in course)

A native API is partly a java driver uses native C language library calls to translate JDBC to native client Library.
These drivers are available for Oracle, Sybase, DB2 and other client library based RDBMS.
Although these drivers are faster than Type 1 drivers, these drivers use native code and require additional permissions to work in an Applet.
This driver might need client side database code to connect over the network.

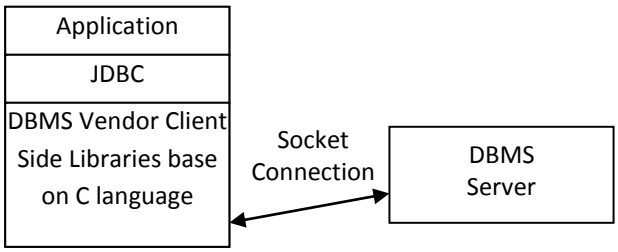

Fig. Native API partly Driver

JDBC-Net Pure Java Driver (Type 3 Drivers) (not in course

This Driver consists of JDBC and DBMS independent protocol driver.
Here the calls are translate and sent to middle tier server through the socket.
The middle tier contacts the databases.
It is simple and most flexible.
It calls the database API on the server.
JDBC requests from the client are first proxied to the JDBC Driver on the server to run.
It and Type 4 drivers can be used by thin (java based) clients as they need no native code.

Native protocol pure java Driver (Type 4 Drivers) (not in course)

It contains JDBC calls that are converted directly to the network protocol used by the DBMS server.
The highest level of driver re-implements the database network API in the Java language.
It can also be used on thin clients as they also have native code.
The network protocol is defined by the vendor and is typically proprietary; the driver usually comes only from the database vendor.

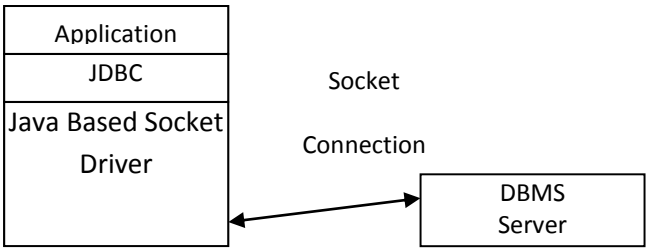

Fig. Native protocol all java driver

The Statement Object

The statement object is created by calling the createStatement( ) method of the connection object.
The statement object is used to execute simple Queries.
It has three methods that can be used for the purpose of querying.

- executeQuery( )
  This is used to execute simple select quer6y and return a single ResultSet object.
- executeUpdate( )
  This is used to execute SQL Insert, Update and Delete statements
- execute( )
  It is used to execute SQL statements that may return multiple values.

In the example programmes we find import the java.sql package.
The JDBC-ODBC bridge is a bridge driver that is loaded by Class.forName( ) method.
The connection object is initialized by getConnection( ) method.
Then, statement is created. Next we execute a simple select query using the executeQuery ( ) method of the statement object (st).

Working with ResultSet

When a database query is executed, the results of the query are returned as a table of data organized according to rows and columns.
The ResultSet interface is to provide access to table data.
Initially the pointer is positioned before the first row.
So we use the next ( ) method to move the cursor to the first row.
You can access the data from the ResultSet rows by calling the getXXX methods.
The XXX is the data type of the parameter such as String, Int and so on.
The column is a ResultSet are accessed beginning with index 1, not 0.
If you specify 0 as the index, you will get runtime error message such as Invalid descriptor Index.
The next program illustrate the executeUpdate( ) method and execute insert, select, update and delete SQL statements.

Obtaining a Connection

The java.sql.Driver class acts as a pass-through driver by forwarding JDBC requests to the real database JDBC Driver.
The JDBC driver class is loaded with a call to Class.forName(drivername).
These next lines of code show how to load two different JDBC driver classes:
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Class.forName("sun.jdbc.odbc.OracleDriver");
Each JDBC driver is configured to understand a specific URL so multiple JDBC drivers can be loaded at one time.
When you specify a URL at connect time, the first machine JDBC driver is selected.
The standard way to establish a connection with a database is to call the method DriveManager.getConnection(URL), the argument URL is a string that provides a way of identifying database.

It has **three parts**:
Syntax:
<protocol>:<subprotocol>:<subname>

Protocol

This is always **jdbc** in the URL.

Subprotocol

This is used for specifying the type of driver or the database connectivity mechanism.
If the mechanism of retrieving the data is JDBC-ODBC Bridge, the subprotocol must be odbc. This is reserved for URLs that specify ODBC style data source names.

Subname

This is used to identify the database.
This is to give enough information to locate the database.

Ex. jdbc:odbc:mydsn

In this example the odbc is the subprotocol and mydns is the local odbc data source name.

Ex. jdbc:oracle:sales

In this example the oracle is the database driver and sales is the local database.

The Connection Object

It represent a connection with the database.
You may have several connection object in an application that connects one or more database.
A database connection can be established with a call to the DriverManager.getConnection method.
The call takes a URL that identifies the database and optionally, the database login user name and password.

```
Connection con=DriverManager.getConnection(url);
Connection con=DriverManager.getConnection(url,"Username","Password");
```

After a connection is established, a statement can be run against the database.
The results of the statement can be retrieved and the connection closed.
Example

```
Connection con=DriverManager.getConnection("jdbc:odbc:mydsn");
Connection con=DriverManager.getConnection("jdbc:oracle","scott","tiger");
```

DriverManager

The DriverManager class is the management layer of JDBC, working between the user and the drivers.
It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver.
In addition, the DriverManager class attends to things like driver login time limits and the printing of log and tracing messages.
For simple applications, the only method in this class that a general programmer needs to use directly is DriverManager.getConnection.
As its name implies, this method establishes a connection to a database.
JDBC allows the user to call the DriverManager methods getDriver, getDrivers, and registerDriver as well as the Driver method connect, but in most cases it is better to let the DriverManager class manage the details of establishing a connection.

PreparedStatement

The PreparedStatement interface inherits from Statement and differs from it in two ways:

1. Instances of PreparedStatement contain an SQL statement that has already been compiled. This is what makes a statement "prepared."
2. The SQL statement contained in a PreparedStatement object may have one or more IN parameters. An IN parameter is a parameter whose value is not specified when the SQL statement is created. Instead the statement has a question mark ("?") as a placeholder for each IN parameter. A value for each question mark must be supplied by the appropriate setXXX method before the statement is executed.

Because PreparedStatement objects are precompiled, their execution can be faster than that of Statement objects. Consequently, an SQL statement that is executed many times is often created as a PreparedStatement object to increase efficiency.

Being a subclass of Statement, PreparedStatement inherits all the functionality of Statement.
In addition, it adds a whole set of methods which are needed for setting the values to be sent to the database in place of the placeholders for IN parameters.
Also, the three methods execute, executeQuery, and executeUpdate are modified so that they take no argument.
The Statement forms of these methods (the forms that take an SQL statement parameter) should never be used with a PreparedStatement object.

Callable Statement

A CallableStatement object provides a way to call stored procedures in a standard way for all DBMSs.
A stored procedure is stored in a database; the *call* to the stored procedure is what a CallableStatement object contains.
This call is written in an escape syntax that may take one of two forms: one form with a result parameter, and the other without one.
(See Section 4, "Statement," for information on escape syntax.) A result parameter, a kind of OUT parameter, is the return value for the stored procedure.
Both forms may have a variable number of parameters used for input (IN parameters), output (OUT parameters), or both (INOUT parameters).
A question mark serves as a placeholder for a parameter.

Example to find the data type of the field in database

```
String query = "select * from Table1";
   ResultSet rs = stmt.executeQuery(query);
   ResultSetMetaData rsmd = rs.getMetaData( );
   int columnCount = rsmd.getColumnCount( );
   for (int i = 1; i <= columnCount; i++)  {
     String s = rsmd.getColumnTypeName(i);
     System.out.println ("Column " + i + " is type " + s);
   }
```

JDBC-ODBC Driver Bridge Driver

If possible, use a Pure Java JDBC driver instead of the Bridge and an ODBC driver.
This completely eliminates the client configuration required by ODBC.
It also eliminates the potential that the Java VM could be corrupted by an error in the native code brought in by the Bridge (that is, the Bridge native library, the ODBC driver manager library, the ODBC driver library, and the database client library).

What Is the JDBC-ODBC Bridge?

The JDBC-ODBC Bridge is a JDBC driver which implements JDBC operations by translating them into ODBC operations.
To ODBC it appears as a normal application program.
The Bridge implements JDBC for any database for which an ODBC driver is available.
The Bridge is implemented as the sun.jdbc.odbc Java package and contains a native library used to access ODBC.
The Bridge is a joint development of Intersolv and JavaSoft.
The Bridge is implemented in Java and uses Java native methods to call ODBC.

The ResultSetMetaData Interface

Consider the following situation suppose you don't know about the column name of the table, Number of columns and their data type.
You cannot access the data through the getXXX ( ) method.
Overcome this problem by ResultSetMetaData interface.
It provides constants and methods that are used to obtain the information about ResultSet object.

getColumnCount ( ) : It returns number of columns in the table of data accessed by a result set.
getColumnName ( ): It returns the name of each column in the database from which the data is retrieved.

Sample Java Program

```
/* Before continuing to this program following requirement is needed

There should be a database with thable emp

Which contains fields id (int) and empName (varchar)

Then you should create System DSN ing ODBC pointing to above database. For simplicity you can create an Access database and create DSN with name mydsn.

*/

import java.sql.*;

public class jdbcDemo
        {


        public static void main(String args[ ])
        {
                ResultSet r;
                try
                {


                        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                        Connection c=DriverManager.getConnection("jdbc:odbc:mydsn");
                        //our data source name is std_dns
                        Statement st=c.createStatement( );
                        r=st.executeQuery("select * from emp");
                        System.out.println("ID\tName");
                        System.out.println("********************");
                        while(r.next( ))
                        System.out.println(r.getString(1)+"\t"+r.getString(2));
                }catch(SQLException e)
                {
                        System.out.println("SQL Error:"+e);
                }
                catch(Exception e)
```

```
                    {
                        System.out.println("Error:"+e);
                    }
                }
        }
```

Sample Java Program

```
import java.io.*;
import java.sql.*;
public class jdbc_opr
{
        static Connection c;
        public static void main(String args[ ]) throws Exception
        {

            int ch;
            /* For Mysql
            Class.forName("com.mysql.jdbc.Driver").newInstance( );
c = DriverManager.getConnection("jdbc:mysql://localhost/cba?user=shiba&password=pword");
        */
            /* For Oracle Uncomment it
            DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver( ));
            c=DriverManager.getConnection("jdbc:oracle:oci8:@cba","scott","tiger");
            */
            //Delete the line below for Oracle and MySql
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            InputStreamReader i=new InputStreamReader(System.in);
            BufferedReader b=new BufferedReader(i);
            try
            {
                //Delete the line below for Oracle and Mysql
                c=DriverManager.getConnection("jdbc:odbc:mydsn");
```

```
do
{
    System.out.println("\t\t\tMenu");
    System.out.println("\t\t\t1. View");
    System.out.println("\t\t\t2. Clear");
    System.out.println("\t\t\t3. Insert");
    System.out.println("\t\t\t4. Modify");
    System.out.println("\t\t\t5. Delete");
    System.out.println("\t\t\t6.exit");
    System.out.println("\t\t\tEnter Your Choice:");

    ch=Integer.parseInt(b.readLine( ));
    switch (ch)
    {
    case 1:
    {
        display( ); break;
    }
    case 2:
    {
        System.out.clrscr( );break;
    }
    case 3:
    {
        insert( );
        display( );    break;
    }
    case 4:
    {
        modify( );
```

```
                    display( );    break;
                }
                case 5:
                {
                        delete( );display( );break;

                }
                case 6:
                {
                        System.out.println("Thank you");
                        System.exit(0);
                }
                default:
                    System.out.println("Invalid Choice");
                }
            }while(ch!=6);
        }
        catch(Exception e){ }
    }
    static void display( )
    {
        try
        {
            ResultSet r;
            int row=0;
            Statement st=c.createStatement( );
            r=st.executeQuery("select * from emp");
            System.out.println("Id\tName");
            System.out.println("**********");
```

```
        while (r.next( ))
            System.out.println(r.getInt(1)+"\t"+r.getString(2));
        r.close( );
    }
    catch(Exception e){ }
    }
    static void clrscr( )
    {
        for(int k=1;k<25;k++)
            System.out.println( );
    }
    static void insert ( )
    {
        try
        {
            int row;
            Statement ins_st;
            ins_st=c.createStatement( );
            System.out.println("4032,shiba");
            row=ins_st.executeUpdate("insert into emp values('4032','shiba')");
            c.commit( );
            System.out.println("No of Rows inserted="+row);
        }
        catch(SQLException e)
        {
                System.out.println("Error in insert"+e);
            }
    }
    static void modify( )
    {
```

```
        try
        {
            int row;
            Statement modi_st;
            modi_st=c.createStatement( );
            row=modi_st.executeUpdate("update emp set name='cba' where name='shiba'");
            c.commit( );
            System.out.println("No of Rows updated="+row);
        }catch(SQLException e)
        {
        }
    }
    static void delete( )
    {
        try
        {
            System.out.println("4032,cba");
            int row;
            Statement del_st;
            del_st=c.createStatement( );
            row=del_st.executeUpdate("delete from emp where id=4032");
            if (row>0)
            {
                c.commit( );
            }
            else
            {
                System.out.println("Record not found");
            }
        }
```

```
                    catch(SQLException e)
                    {
                        System.out.println("Error in delete" +e);
                    }
                }
        }
/*
        Note: System.out.clrscr( ) is removed in new version of jdk.
*/
```

Reference: *Java Documentation from sum.com*

Java Tutorial from sun.com

Oracle Handout (SSI book) and Complete Reference of Oracle 9i, Oracle Press

Internet and Java programming by R. Krishnamoorthy and S. Prabhu

Extra Stuff

"Why do you need JDBC but not ODBC?" There are several answers to this question:

1. ODBC is not appropriate for direct use from Java because it uses a C interface. Calls from Java to native C code have a number of drawbacks in the security, implementation, robustness, and automatic portability of applications.
2. A literal translation of the ODBC C API into a Java API would not be desirable. For example, Java has no pointers, and ODBC makes copious use of them, including the notoriously error-prone generic pointer "void *". You can think of JDBC as ODBC translated into an object-oriented interface that is natural for Java programmers.
3. ODBC is hard to learn. It mixes simple and advanced features together, and it has complex options even for simple queries. JDBC, on the other hand, was designed to keep simple things simple while allowing more advanced capabilities where required.
4. A Java API like JDBC is needed in order to enable a "pure Java" solution. When ODBC is used, the ODBC driver manager and drivers must be manually installed on every client machine. When the JDBC driver is written completely in Java, however, JDBC code is automatically installable, portable, and secure on all Java platforms from network computers to mainframes.

Why Java is recommended to use Oracle and not to use Ms-Access as backend database?

Oracle	Ms-Access
Java has its own Oracle JDBC Driver	Java doesn't have special Ms-Access JDBC Driver and use JDBC-ODBC bridge driver.
Oracle JDBC driver is included in the Server Application	Since it uses JDBC-ODBC bridge driver it don't have its driver on Server Application and ODBC should be configured in each client.
Oracle Provides concurrently use of Database	Ms-Access doesn't provide concurrent access to the database.

Since it has its own JDBC driver the performance is high and database access if fast.	Since JDBC-ODBC bridge driver, its performance is low as ODBC configuration has to be configured support JDBC. So, database access is also slow.
Java is most widely used for Distributed or Network base Application like web, Network Applications so, Oracle Database is preferred as it is a Server based Database. So we don't have to install Oracle in every client.	Ms-Access is not a Server base Database so it is not preferred for Distributed database.
Oracle database also have embedded SQL for Java called SQLJ	Ms-Access Don't has embedded SQL for Java
Oracle corporate also have developed Java for Oracle and compilers for them	Microsoft has not developed the Java for Ms-Access
It uses Pure Native Jave JDBC driver	It doesn't have.

Statement Versus PreparedStatement

Statement	PreparedStatement
1. It is a super class of PreparedStatement class.	1. It is a subclass of Statement class.
2. Instances of Statement contain an SQL statement that has not been compiled.	2. Instances of PreparedStatement contain an SQL statement that has already been compiled. This is what makes a statement "prepared."
3. We can not use IN parameter in SQL Statement.	3. We can use IN parameter in SQL Statement.
4. It consists of less method and properties.	4. As it is the subclass of Statement, it consists of all most all the methods and properties of the Statement class and it has its additional properties and methods.
5. It is slower to execute than PreparedStatement	5. It is relatively faster
6. It has to do verify its metadata against the database many times.	6. It has to verify its metadata only once
7. Under typical use of SQL statement objects performance is better than PreparedStatement where there are small number of SQL statements	7. Under typical use of SQL statement objects performance is worse than Statement when there is small number of SQL statements
8. It is not design to work with large objects like BLOBs and CLOBs	8. It is design to work with large objects like BLOBs and CLOBs